Traduction dirigée par la syntaxe

Christophe Alias

Objectif

- Production de code directe à partir du programme
- On produit du pseudo-code pour un modèle de machine (pile ou registre).
- Plus tard:
  - Optimisation du pseudo-code
  - Production de “vrai” code machine x86-64

Langage d'entrée

P ::= D S … S

D ::= int id, …, id;

S ::= x = E;

| if(C) S else S

| while (C) do S

| {S … S}

| RT

E ::= cst | id | E op E

C ::= E>E

RT ::= input(id); | output(id);

```
int x,y;
input(x);
if(x>0) y = x;
else y = -1*x;
output(y);
```

Machine à pile

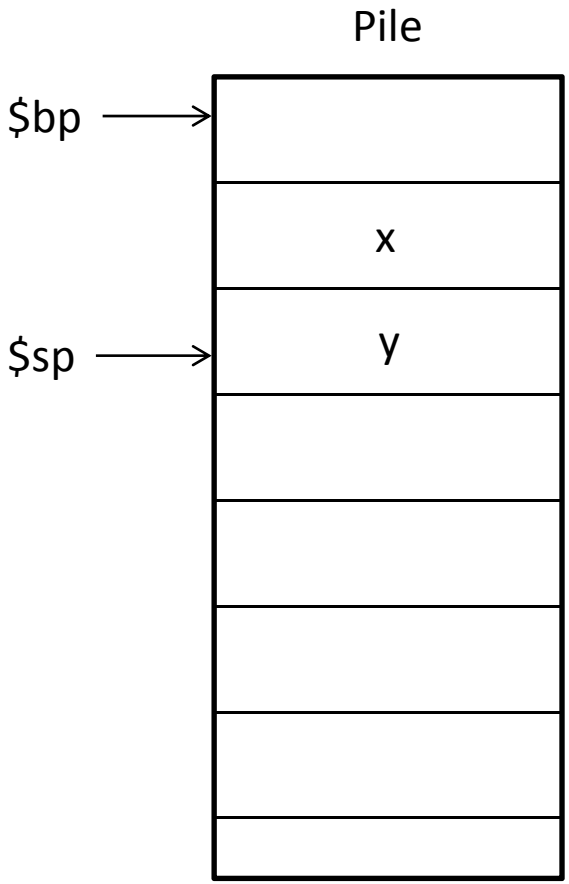
Les données sont dans une pile

$bp: fond de pile

$sp: sommet de pile

$\delta(x) = 0$

$\delta(y) = 1$



Machine à pile

Alloc n

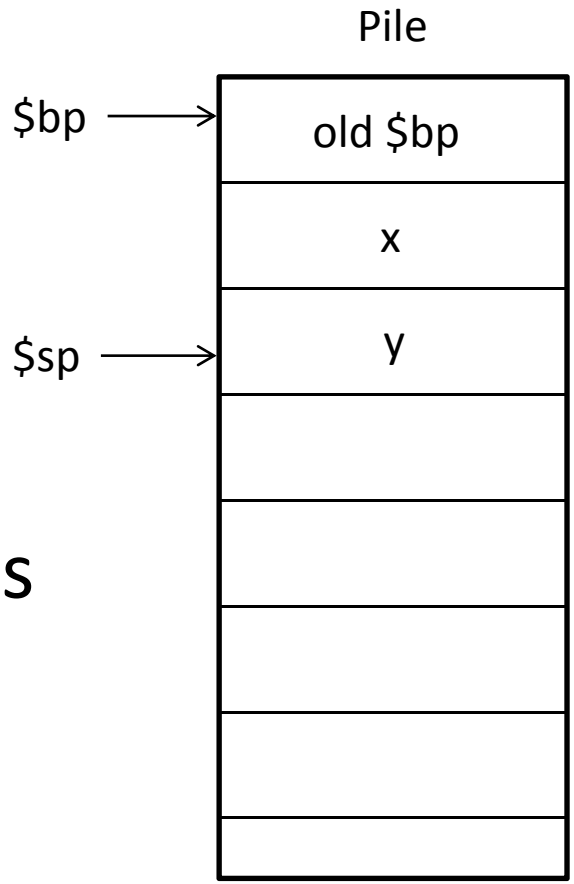
Sauvegarde $bp

Alloue n slots de pile

Free

Libère les derniers slots alloués

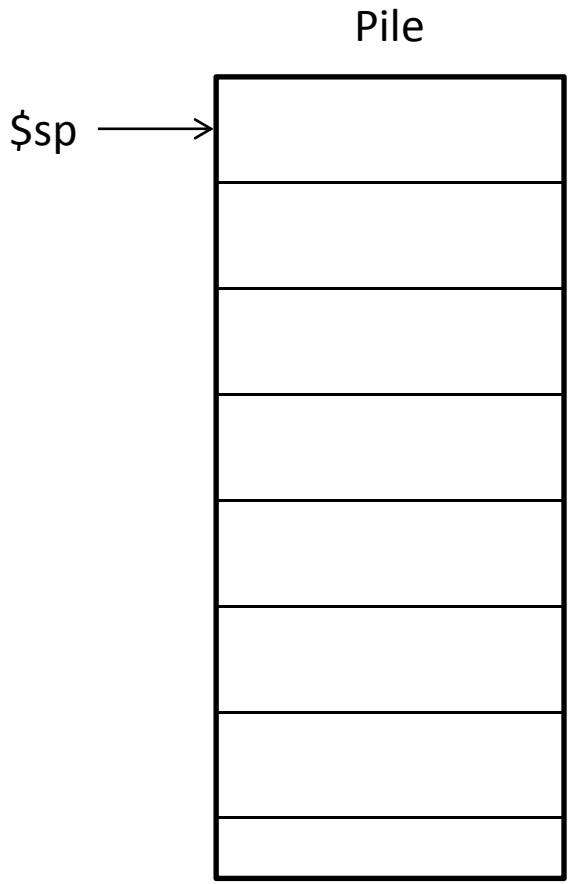
Restore $bp



Quizz

Dérouler l'exécution de:

Alloc 2
Alloc 1
Free



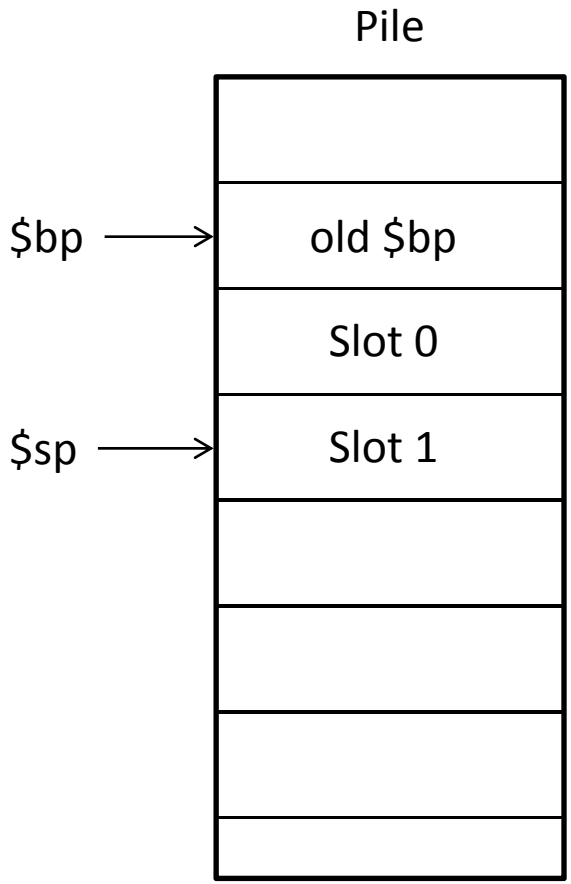
Quizz

Dérouler l'exécution de:

Alloc 2

Alloc 1

Free



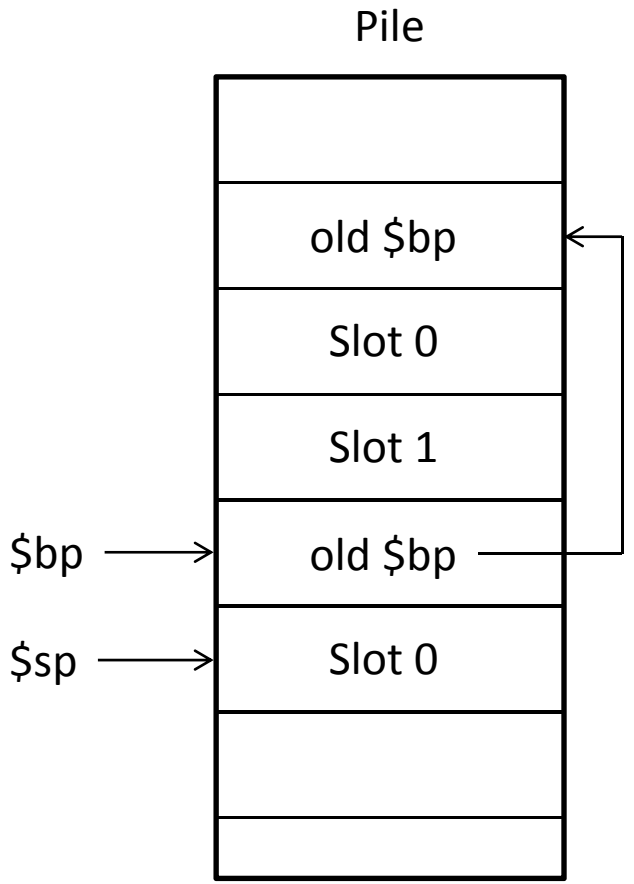
Quizz

Dérouler l'exécution de:

Alloc 2

Alloc 1

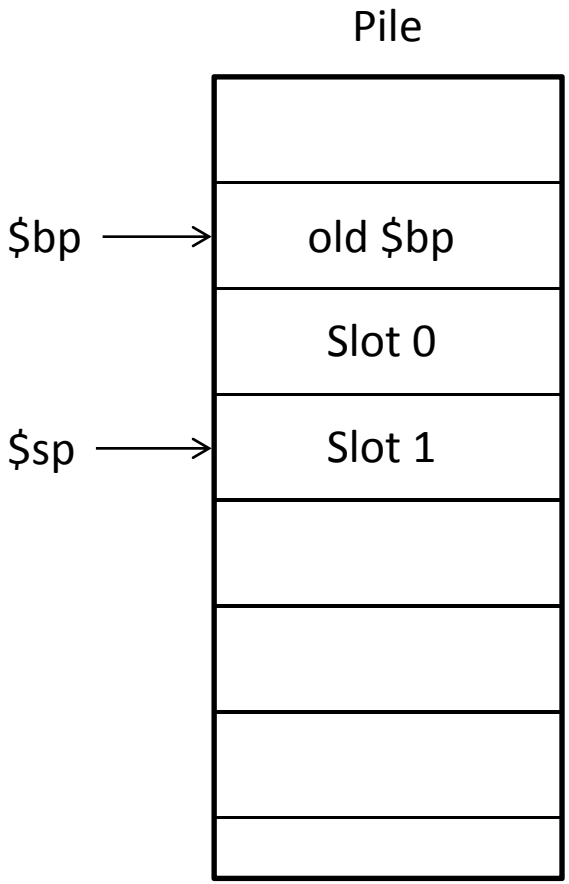
Free



Quizz

Dérouler l'exécution de:

Alloc 2
Alloc 1
Free



Entrées/sorties

push #slot

empiler(#slot)

… ->

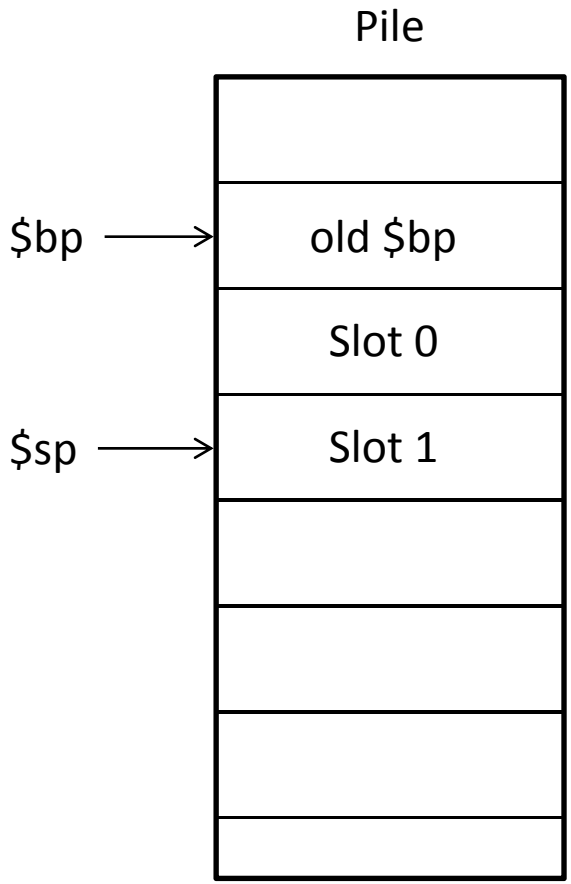
… slot_value ->

pop #slot

dépiler(#slot)

… value ->

… ->



Opérations arithmétiques

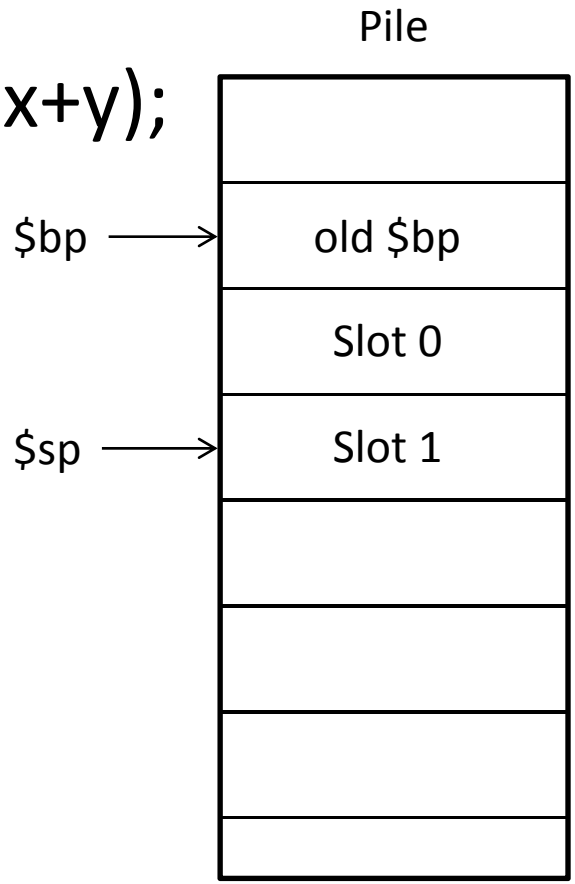
add

dépiler(y); dépiler(x); empiler(x+y);

… x y ->

… x+y ->

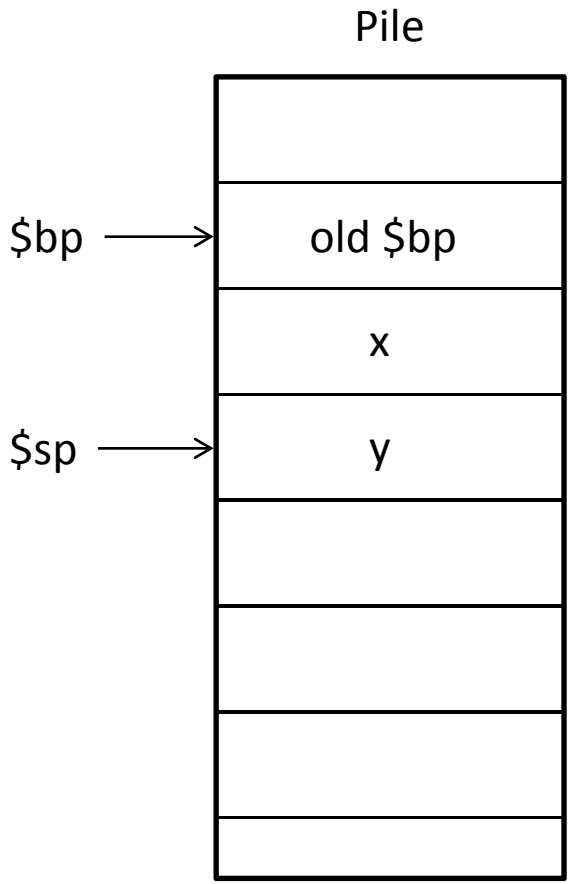
sub, mul, div



Quizz

Que fait le programme:

```
push 0
push 1
add
pop 0
```



Quizz

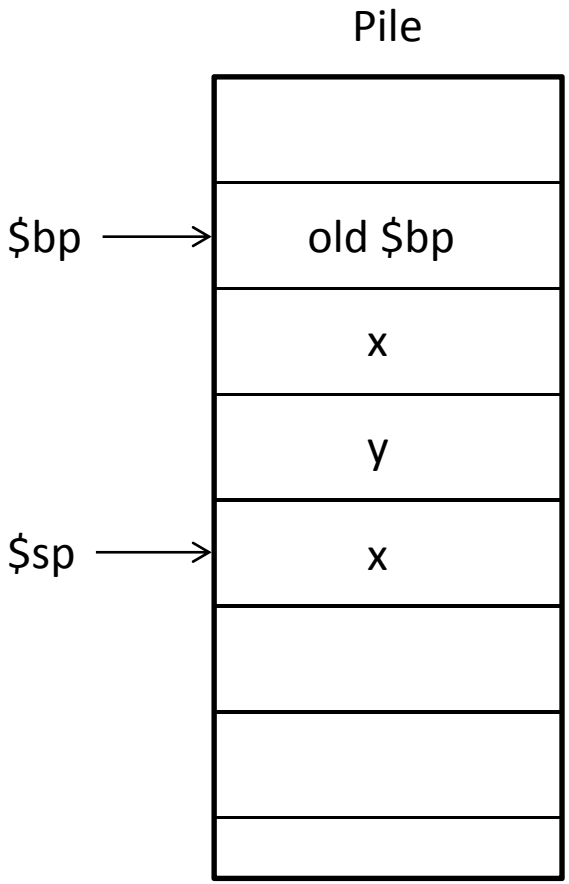
Que fait le programme:

push 0

push 1

add

pop 0



Quizz

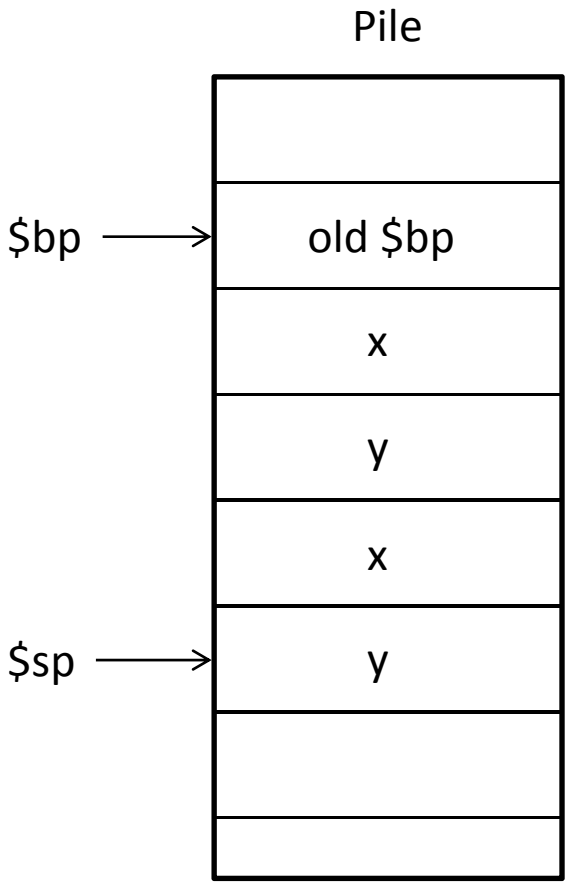
Que fait le programme:

push 0

push 1

add

pop 0



Quizz

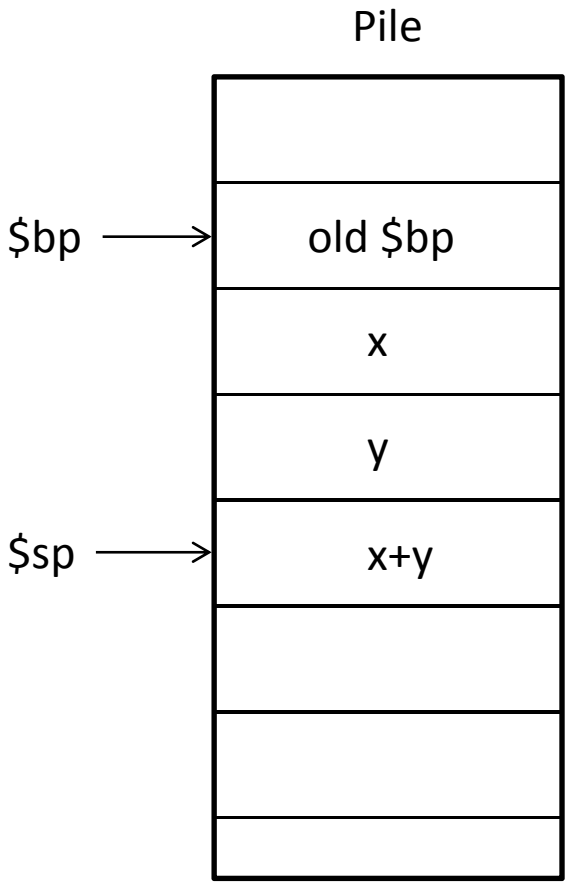
Que fait le programme:

push 0

push 1

add

pop 0



Quizz

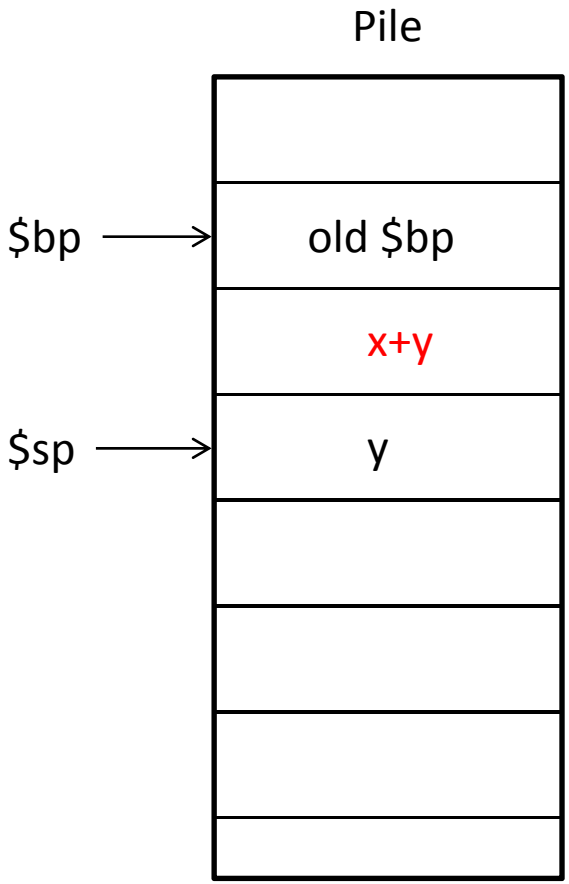
Que fait le programme:

push 0

push 1

add

pop 0



Valeurs immédiates

mpush imm

… ->

… imm ->

Tests (if, while)

testg

dépiler(y); dépiler(x); empiler(x>y);

true = 1, false = 0

… x y ->

… x>y ->

testl, teste, testne

Sauts (if, while)

jmp @

branche à l'adresse @

jz @

dépiler(x)

si x = 0 alors jmp @

... x ->

... ->

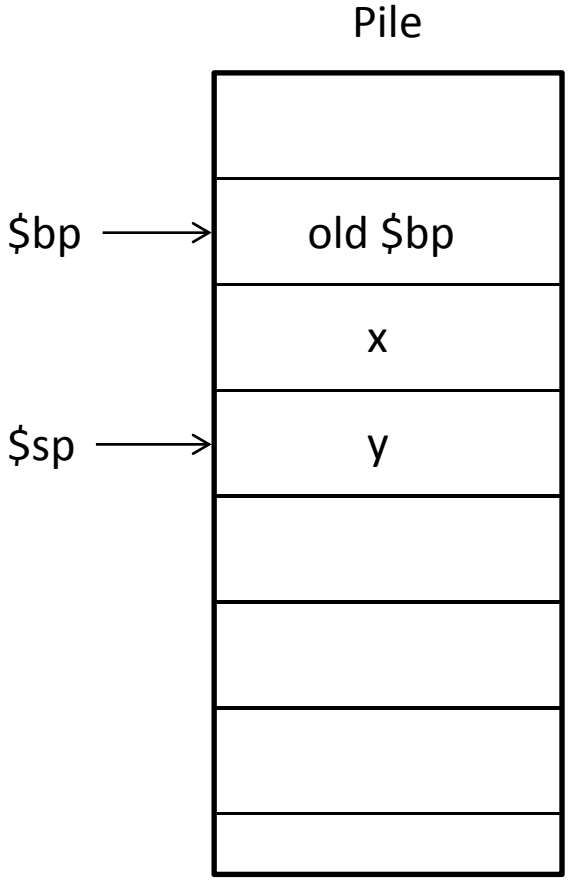
Quizz

Que fait le programme:

```
push 0
mpush 0
testl
jz end
mpush -1
push 0
mul
pop 1
end:
```

if(x<0)

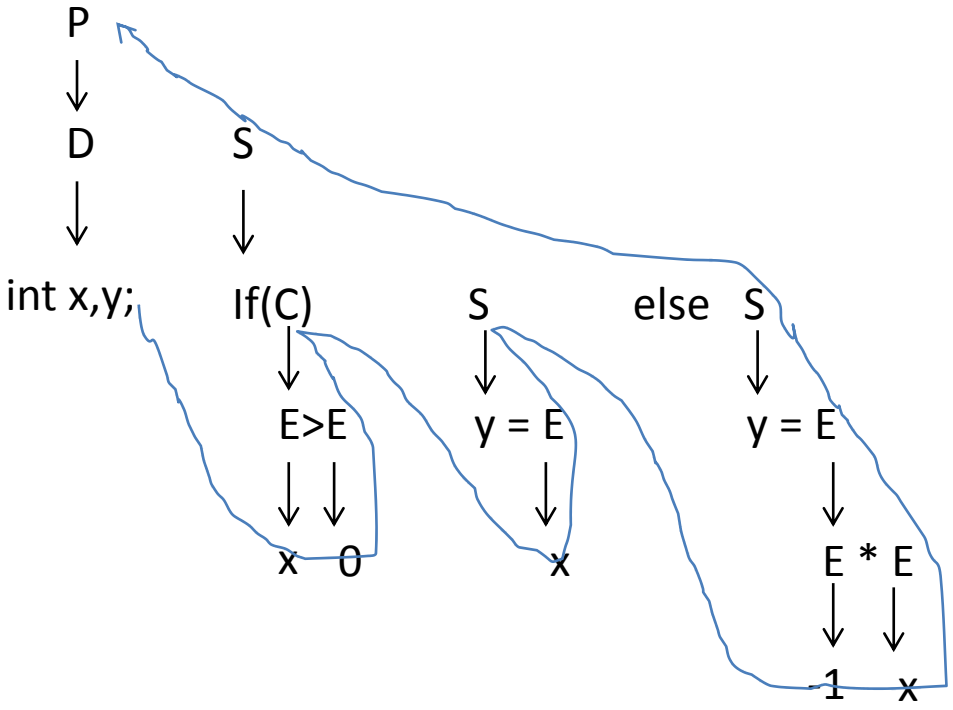
y = -1*x;



Traduction dirigée par la syntaxe

- On parcours l'arbre de dérivation du programme dans l'ordre postfixe
- On produit du code pour chaque dérivation

```
int x,y;
if(x>0) y = x;
else y = -1*x
```



Règles de traduction

- ⟦E⟧ est le code pile qui évalue E et place le résultat sur la pile:

  … ->

  … E ->

- ⟦x+1⟧ =

  push δ(x)

  mpush 1

  add

- On défini ⟦.⟧ pour chaque dérivation

Traduction des expressions

E ::= cst | id | E op E

- ⟦cst⟧ =
  mpush cst
- ⟦id⟧ =
  push δ(id)
- ⟦E1 + E2⟧ =
  ⟦E1⟧
  ⟦E2⟧
  add

Quizz

Donner $[\![x+1]\!]$ avec $\delta(x) = 0$

$[\![x+1]\!]$ =

$[\![x]\!]$ — push $\delta(x)$

$[\![1]\!]$ — mpush 1

add

Traduction des Tests

C ::= E>E

⟦E1>E2⟧ =
⟦E1⟧
⟦E2⟧
testg

Traduction du contrôle

S ::= x = E; | if(C) S else S | while (C) do S
| {S … S} | RT

- ⟦x = E⟧ = ⟦E⟧ pop δ(x)
- ⟦if(C) S1 else S2⟧ =

⟦C⟧
jz else
⟦S1⟧
jmp endif
else:
⟦S2⟧
endif:

Traduction du programme

- P ::= D S … S
- ⟦int id1, …, idn; S1 … Sp⟧ =
  alloc n
  ⟦S1⟧ … ⟦Sp⟧
  free
  stop

Implémentation en C++

Parcours récursif de l'arbre de dérivation:

La règle: ⟦E1+E2⟧ = ⟦E1⟧ ⟦E2⟧ add

s'écrit:

```
void codegen(AstNode* node) {
  if(node->is_add()) {
    codegen(node->left());
    codegen(node->right());
    add();
  }
  …
}
```

Implémentation avec Yacc

La règle:

$[\![E1+E2]\!] = [\![E1]\!]\,[\![E2]\!]$ add

S'implémente en Yacc:

```
E: E TK_PLUS E { add(); } ;
```

Ce qui est équivalent au programme C++